CPU VIRTUALIZATION

Cheuk on, CHUNG
Hsiang yu Cheng
Challenges without VT-x

• Techniques such as paravirtualization and dynamic binary translation faced some hard technical challenges in virtualizing the existing x86 architecture
  • Ring aliasing and compression
  • Address space compression
  • Non-faulting access to privileged state
  • Unnecessary impact of guest transitions
    • Suspend interrupts within critical regions
  • Interrupt virtualization
  • Access to hidden state
VT-x Architecture: Root and non-root mode

- The processor is at any time either in root mode or in non-root mode
- Duplicates the entire architecturally visible state of the processor
- **Atomic** transition
  - One instruction to do transition
- Only use for virtualization, orthogonal to all other modes on CPU
VT-x Architecture:
Root and non-root mode

• Follows Popek/Goldberg criteria
  • With root-mode privileged instructions

• Orthogonal to protection levels of protected mode with all 4 levels separately available to each mode
  • Solves: Ring aliasing and compression

• Each mode defines its own distinct complete 64-bit linear address space, with distinct page table tree and CR3
  • Solves: Address space compression

• Each mode has its own interrupt flag
  • Solves: Interrupt virtualization
Root mode

• **Hypervisor** and **host operating system**

• Can only be detected by executing specific new instruction only available in root mode
  • Guest can not know if it is virtualized
Non-root mode

• **Virtual machines**

• All root-mode-privileged instructions are either:
  • Implemented by the processor and operate exclusively on the non-root duplicate of the processor
  • Or cause a trap

• Executing instruction in hardware has better performance
  • Tradeoff between hardware complexity and overall performance

• Grant full access to 4 level rings to guest OS
  • Solves: Ring aliasing and compression
VT-x and Popek/Goldberg Theorem

• Does not take into account whether instructions are privileged or not, instead only takes into consideration the orthogonal question of whether they are root-mode privileged

• However, reducing transitions by implementing certain sensitive instructions in hardware is necessary to meet the performance criteria.
• Guest OS running on non-root-%cpl=0
  • Its privileged instructions operate on the non-root duplicate of the processor state.
• Possible ways
  • Execute instructions in hardware without informing the hypervisor
  • Generate a trap for hypervisor to emulate the instruction
Book examples - 2

• Sensitive instructions available in usermode
  • cli, sti, popf, pushf
  • Are oftenly used in modern OS; thus, handled directly by the processor

• Behavior-sensitive instructions available in usermode
  • sgdt, sidt
  • Not very common; therefore are Root-mode-privileged, and cause trap when executed
VT-x Control Flow

- Virtual Machine Control Structure (VMCS)
  - In physical memory
  - Layout is undefined
  - Access with `#vmread`, `#vmwrite`
- `#vmlaunch`, `#vmresume`
  - Enter non-root mode
- `#vmexit`
  - Non-root to root
  - Comes with a trap
  - `vmcs.exit_reason`
VMCS: Exit Reasons - 1

• Stored in register: `vmcs.exit_reason`
• Any attempt by guest to execute a root-mode-privileged instruction
  • Most of the privileged instructions
  • Sensitive-yet-unprivileged instructions
• `vmcall` made by guest operating system
• Exceptions result from executing innocuous instruction in non-root mode, that happens to take a trap
  • Page faults caused by shadow paging and access to memory-mapped I/O
  • General-purpose faults due to segment violations
VMCS: Exit Reasons - 2

• EPT violations
  • Subset of page faults caused when the extended page mapping (Chapter 5)

• External interrupts that occurred while the CPU is executing in non-root mode
  • Network or disk I/O
  • Must be handled by hypervisor or host OS

• Interrupt window opens and the virtual machine is pending interrupt
  • Following #vmexit, the hypervisor can emulate the interrupt for vm

• ISA extensions introduced with VT-x
  • Doesn’t occur in “normal virtualization”, but important in nested virtualization
<table>
<thead>
<tr>
<th>Category</th>
<th>Exit Reason</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception</td>
<td>0</td>
<td>Any guest instruction that causes an exception</td>
</tr>
<tr>
<td>Interrupt</td>
<td>1</td>
<td>The exit is due to an external I/O interrupt</td>
</tr>
<tr>
<td>Triple fault</td>
<td>2</td>
<td>Reset condition (bad)</td>
</tr>
<tr>
<td>Interrupt window</td>
<td>7</td>
<td>The guest can now handle a pending guest interrupt</td>
</tr>
<tr>
<td>Legacy emulation</td>
<td>9</td>
<td>Instruction is not implemented in non-root mode; software expected to provide backward compatibility, e.g., task switch</td>
</tr>
<tr>
<td>Root-mode Sensitive</td>
<td>11-17, 28-29, 31-32, 46-47:</td>
<td>x86 privileged or sensitive instructions: getsec, hlt, invd, invlpg, rdpmc, rdtsc, rsm, mov-cr, mov-dr, rdmsr, wrmsr, monitor, pause, lgdt, lidt, sgdt, sidt, lldt, ltr, slidt</td>
</tr>
<tr>
<td>Hypercall</td>
<td>18</td>
<td>vmcall: Explicit transition from non-root to root mode</td>
</tr>
<tr>
<td>VT-x new</td>
<td>19-27, 50, 53</td>
<td>ISA extensions to control non-root execution: invect, invvpid, vmclear, vmlaunch, vmptrld, vmptrst, vmreas, vmresume, vmwrite, vmxoff, vmxon</td>
</tr>
<tr>
<td>I/O</td>
<td>30</td>
<td>Legacy I/O instructions</td>
</tr>
<tr>
<td>EPT</td>
<td>48-49</td>
<td>EPT violations and misconfigurations</td>
</tr>
</tbody>
</table>
Virtualizing the CPU and ignore the MMU

• Root-mode and non-root made each has a distinct CR3 register.
  • Switched as part of \texttt{vmentry} and \texttt{#vmexit}
  • 100\% disjoint address spaces
• Other aspect of memory virtualization is left to software
• Two possible approaches
  • Shadow the page tables of the virtual machine with host-physical values
  • Rely on paravirtualization of the virtual memory subsystem and inform the hypervisor to validate all new mappings
Virtualizing the CPU and ignore the MMU

• VMWare: *Shadow paging*

• Bad performance
  • Over 90% of #vmexit were due to shadow paging
  • Disabling VT-x and use software techniques is even faster

• All guest’s page table pages are downgraded to read-only mappings to ensure a trap, or #vmexit in VT-x

• A hypervisor that relies entirely on direct execution will therefore suffer a trap for every change in the virtual memory of guest OS
Virtualizing the CPU and ignore the MMU

• VMWare: *Adaptive dynamic binary translation*
  • Relies on locality of instructions that manipulate page table entries in an OS
  • Identifies the locations of those instructions and emulate them.
  • Update shadow entries without dereferencing the memory location
  • Avoid taking expensive traps

• This anomaly was addressed in subsequent processors
  • With architectural support to MMU virtualization (Chapter 5)
Case study: KVM

KVM module
1. kernel module in linux, act as a hypervisor
Each guest os is a process, each vcpu is a thread.
Guest OSs is managed by linux existing module

2. virtualize the cpu
initialize the vcpu by setting register
(e.g. cr4 for virtual mode)
switch cpu mode

QEMU-KVM
virtualize I/O(hard disk/network access)
CPU Mode

Three mode

non root mode (guest mode): execute normal (neither privileged nor sensitive) instructions

User mode: for i/o emulation, executed by qemu

Kernel mode: execute sensitive or privileged instruction (e.g. SGDT, SIDT).
VMExit is triggered by sensitive/privileged instruction then cpu switch mode from non-root to kernel mode,

If it is I/O instruction, cpu switch from kernel mode to user mode. Then transfer the control to qemu. Then qemu will emulate the I/O instruction.
Within the KVM kernel module

• Restores the current state of the virtual CPU
• Enters non-root mode using `#vmresume`
• Virtual machine executes until next `#vmexit`
• Handles the `#vmexit` according to exit reason
• If the guest issued a programmed I/O operation(`exit_reason = IO`) or a memory-mapped I/O instruction(`exit_reason = exception`, only when accessing a memory-mapped I/O page), return to user mode
• If the `#vmexit` is caused by an external event, return to userspace
Control Flow of Handling VMexit

**Fetch** the instruction from guest virtual memory
**Decode** the instruction, extracting its operator and operands
**Verify** whether the instruction can execute given the current state of the virtual CPU
**Read** any memory read-operands from memory
**Emulate** the decoded instruction
**Write** any memory write-operands back to the guest virtual memory
**Update** guest registers and the instruction pointer as needed
Questions